(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
19 April 2001 (19.04.2001)

PCT

(10) International Publication Number
WO 01/27751 A1

(51) International Patent Classification[7]: G06F 9/445

(21) International Application Number: PCT/US00/24649

(22) International Filing Date:
6 September 2000 (06.09.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/417,979 13 October 1999 (13.10.1999) US

(71) Applicant: TRANSMETA CORPORATION [US/US]; 3940 Freedom Circle, Santa Clara, CA 95054 (US).

(72) Inventors: TORVALDS, Linus; 1050 Woodduck Avenue, Santa Clara, CA 95051 (US). ANVIN, H., Peter; 4390 Albany Drive #46, San Jose, CA 95129 (US).

(74) Agent: KING, Stephen, L.; 30 Sweetbay Road, Rancho Palos Verdes, CA 90275 (US).

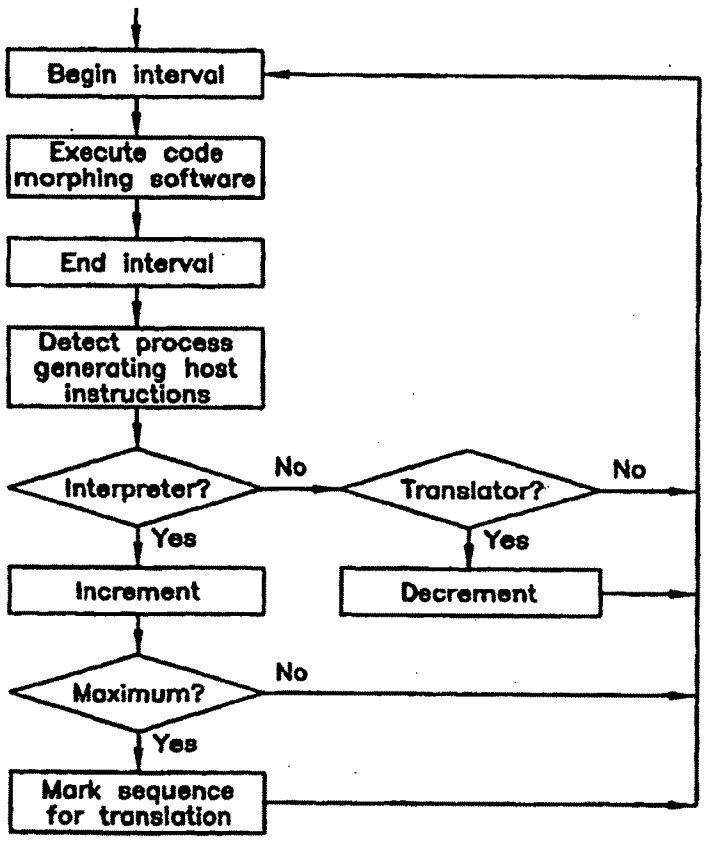
(81) Designated States *(national)*: CA, CN, JP, KR.

(84) Designated States *(regional)*: European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published:
— *With international search report.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: METHOD OF CHANGING MODES OF CODE GENERATION



(57) Abstract: A method for determining a process as shown in figure 1 to use for converting instructions in a target instruction set to instructions in a host instructions set including the steps of executing code morphing software including an interpreter and a translator to generate host instructions from target instructions, detecting at intervals whether the interpreter or the translator is executing, increasing a count if the interpreter is executing and decreasing the count if the translator is executing, and changing from interpreting to translating a sequence of target instructions when the count reaches a selected maximum.

WO 01/27751 A1

METHOD OF CHANGING MODES OF CODE GENERATION

BACKGROUND OF THE INVENTION

Field Of The Invention

This invention relates to computer systems and, more particularly, to methods for increasing the efficiency of operation of a microprocessor which dynamically translates instructions from a target to a host instruction set.

History Of The Prior Art

Recently, a new microprocessor was developed which combines a simple but very fast host processor (called a "morph host") and software (referred to as "code morphing software") to execute application programs designed for a "target" processor having an instruction set different than the instruction set of the morph host processor. The morph host processor executes the code morphing software to translate the application programs into morph host processor instructions which accomplish the purpose of the original target software. As the target instructions are translated, the new host instructions are both executed and stored in a translation buffer where they may be accessed without further translation. Although the initial translation of a program is slow, once translated, many of the steps normally required for hardware to execute a program are eliminated. The new microprocessor has demonstrated that a simple fast processor designed to expend little power is able to execute translated "target" instructions at a rate equivalent to that of the "target" processor for which the programs were designed.

In order to be able to run programs designed for other processors at a rapid rate, the morph host processor includes a number of hardware enhancements. One of these enhancements is a gated store buffer which resides between the host processor and the translation buffer. A second enhancement is a set of host registers (in addition to normal working registers) which store known state of the target processor existing prior to any sequence of target instructions being translated. Memory stores generated as sequences of morph host instructions are executed are placed in the gated store buffer. If the morph host instructions execute without raising an exception, the target state at the beginning of the sequence of instructions is updated to the target state at the point at which the sequence completed and the memory stores are committed to memory.

It will be noted that the method by which the new microprocessor handles the execution of translations by placing the effects generated by execution in temporary storage until execution of the translation has been completed is effectively a very rapid method of speculating. The new microprocessor, in fact, uses the same circuitry for speculating on the outcome of other operations. For example, by temporarily holding the results of execution of instructions reordered by a software scheduler from naively translated instructions, more aggressive reordering may be accomplished than has been attempted by the prior art. When such a reordered sequence of instructions executes to produce a correct result, the memory stores resulting from execution of the reordered sequence may be committed to memory and target state may be updated. If the reordered sequence generates an exception while executing, then the state of the processor may be rolled back to target state at the beginning of the

sequence and a more conservative approach taken in translating the sequence.

One of the most advantageous features of the new microprocessor is its ability to link together long sequences of translated instructions. Once short sequences of target instructions have been translated and found to execute without exception, it is possible to link large numbers of these short sequences together to form long sequences of instructions. This allows a translated program to be executed at great speed because the microprocessor need not go through all of the steps (such as looking up each of the shorter translated sequences) normally taken by hardware processors to execute instructions. Even more speed may be attained than might be expected because, once long sequences are linked, it is often possible for an optimizer to eliminate many of the steps from the long sequences without changing the results produced. Hardware optimizers have never been able to optimize sequences of instructions long enough to allow the patterns which allow significant optimization to become apparent.

A problem which has occurred with the new processor relates to those instructions of the target application which are executed only an insignificant number of times. For example, instructions required to initiate operation of a particular application are often executed only when the application is first called; and instructions required to terminate operation of an application are often executed only when the program is actually terminated. However, the new processor typically treats all instructions in the same manner. It decodes a target instruction, fetches the primitive host instructions which carry out the function for which the target instruction is designed, proceeds through a very extensive process

of optimizing, and then stores the translated and optimized instructions in the translation cache. As the operation of the new processor proceeds, the sequences of translated instructions are linked to one another and further optimized; and the longer sequences of linked instructions are stored in the translation buffer. Ultimately, large blocks of translated instructions are stored as super-blocks of host instructions. When an exception occurs during execution of a particular host instruction or linked set of instructions, the new processor goes through the process of rolling back to the last correct state of the target processor and then provides single-step translations of the target instructions from the point of the last correct state to the point at which the exception again occurs. These translations are also stored in the translation cache. The new processor is described in detail in U.S. patent 5,832,205, Kelly et al., issued November 3, 1998, and assigned to the assignee of the present invention.

Although this process creates code which executes rapidly, the process has a number of effects which limit the overall speed attainable and may cause other undesirable effects. First, the process requires a substantial amount of storage capacity for translated instructions. Many times a number of different translations exist for the same set of target instructions because the sequences were entered from different branches. Once stored, the translated instructions occupy this storage until removed for some affirmative reason. Second, if a sequence of instructions is to be run but once, the time required for translating and optimizing may be significantly greater than the time needed to execute a step-by-step translation of the initial target instructions. This tends to lower the average speed of the new processor.

For these reasons, the original processor was modified to include as a part of the code morphing software, an interpreter which accomplishes step-by-step translation of each of the target instructions. Although there are many possible embodiments, an interpreter essentially fetches a target instruction, decodes the instruction, provides a host process to accomplish the purpose of the target instruction, and executes the host process. When it finishes interpreting and executing one target instruction, the interpreter precedes to the next target instruction. This process essentially single steps through the interpretation and execution of target instructions. As each target instruction is interpreted and executed, the state of the target processor is brought up to date. The host instructions produced by the interpreter are not typically stored in the translation cache so linking and the further optimizations available after linking are not carried out. The interpreter continues this process for the remainder of the sequence of target instructions.

It was determined that, in general, not until some number of executions of any sequence of instructions have occurred does the time required for all of the previous interpretations and executions become equal to the time required to translate and optimize the sequence. Consequently, for instructions which are little used during the execution of an application, it is often desirable to utilize the interpreter instead of the translator software. Thus, a sequence of instructions which runs only once is often better and more rapidly handled by simply interpreting and never translating the sequence.

In order to make use of this advantage, the improved processor was modified to utilize the interpreter whenever a sequence of target instructions is first encountered. The interpreter software is associated

with a counter which keeps track of the number of times sequences of instructions are executed. The interpreter may be run each time the sequence is encountered until it has been executed some number of times without generating an exception. When a target instruction has been interpreted and executed some selected number of times during the particular sequence, the code morphing software switches from the interpreter to the translator and its attendant optimization and storage processes. When this occurs, a sufficient number of executions will have occurred that it is probable that execution of the instructions will reoccur; and a stored optimized translation will provide significantly faster execution of the applications as a whole.

When the code morphing software switches to the normal translation process, the translation is optimized and stored in the translation cache. Thereafter, that translation may be further optimized and linked to other translations so that the very high speeds of execution realized from such processes may be obtained.

An especially useful embodiment of the improved processor records data relating to the number of times a target instruction is executed by the interpreter only at points at which branches occur in the instructions. The interpreter single steps through the various target instructions until a branch occurs. When a branch instruction occurs, statistics regarding that particular branch instruction (the instruction with the particular memory address) are recorded. Since all of the target instructions from the beginning of a sequence until the branch will simply be executed in sequential order, no record need be kept until the point of the branch; and a significant number of steps related to storage in the translation cache are eliminated.

Moreover, if the interpreter is utilized to collect statistics in addition to the number of times a particular target instruction has been executed, additional significant advantages may be obtained. For example, if a target instruction includes a branch, the address of the instruction to which it branches may be recorded along with the number of times the branch has been executed. Then, when a number of sequential target instructions are executed by the interpreter, a history of branching and branch addresses will have been established. These statistics may be utilized to determine whether a particular sequence of instructions is probably going to become a super-block of translated instructions. By utilizing these statistics, a particular sequence of instructions may be speculatively considered to be a super-block after being executed a significant number of times. After being interpreted for the selected number of times, the sequence may be translated, optimized, linked through the various branches without the necessity to go through a separate linking operation, and stored as such in the translation cache. If the speculation turns out to be true, then significant time is saved in processing the instructions. If not, the operation simply causes an exception which returns the code morphing software to the interpreter.

Not only is the interpreter useful for generating host code for sequences which are used infrequently, it is also utilized in handling exceptions. Whenever the modified processor encounters a target exception while executing any translated target application, the code morphing software causes a rollback to occur to the last known correct state of the target processor. Then, the interpreter portion of the code morphing software is utilized rather than the translator portion to provide host instructions. The interpreter single steps through the generation and execution of

target instructions. As each target instruction is interpreted and executed, the state of the target processor is brought up to date.

The interpreter continues this process for the remainder of the sequence of target instructions until the exception again occurs. At this point, the state of the target processor is correct for the state of the interpretation so that the exception can be handled correctly and expeditiously. Because the interpretation process is so simple, the process of determining the point of occurrence of a target exception is significantly faster than the determination of such a point when carried out by the translation process which goes through the above-described translation and optimization process and then is stored in the translation cache.

By combining the interpreter with the optimizing translator which functions as a dynamic compiler of sequences of translated instructions, the code morphing software removes many of the limits to the upper speed of execution of target applications by the new processor. The use of the interpreter to handle early executions of instructions eliminates the need to optimize instructions which are little used during execution of the application and thereby increases the speed of operation. The need to store these little used instructions in the translation cache reduces the need for storage and eliminates the need for discarding many translated instructions. The use of the interpreter to handle exceptions produces the same useful effects as using the translator yet speeds operations and reduces storage requirements.

The improved processor is described in detail in U. S. patent application Serial No. _____, entitled Method For Integration Of Interpretation

And Translation In A Microprocessor, R. Bedichek et al., filed on even date herewith, and assigned to the assignee of the present invention.

Even though the combination of an interpreter and a translator functions to greatly improve the operation of the unique microprocessor, some problems in operation remain. These problems may be generally described as an inability to utilize the two functions optimally. Because there are so many types of operations conducted by sequences of instructions in any application program, it is quite difficult to determine to the point at which interpretation should end and translation begin. Often a process which has been interpreted for a sufficient number of times to be translated is never again used so the code simply occupies space in the translation cache. Other processes are reused constantly. Moving the point at which translation commences does not appear to solve the problem.

It is desirable to improve the operational speed of the improved microprocessor so that it executes more rapidly by modifying the processes for controlling the use of the interpreter and translator software of the code morphing software.

Summary Of The Invention

It is, therefore, an object of the present invention to provide a faster microprocessor compatible with and capable of running application programs and operating systems designed for other microprocessors.

This and other objects of the present invention are realized by a method for determining a process to use for converting instructions in a target instruction set to instructions in a host instructions set comprising the

steps of executing code morphing software including an interpreter and a translator to provide host instructions from target instructions, detecting at intervals whether the interpreter or the translator is executing, increasing a count if the interpreter is executing and decreasing the count if the translator is executing, and changing from interpreting to translating a sequence of target instructions when the count reaches a selected maximum.

These and other objects and features of the invention will be better understood by reference to the detailed description which follows taken together with the drawings in which like elements are referred to by like designations throughout the several views.

Brief Description Of The Drawings

Figures 1 is a flow chart illustrating a first embodiment of the invention.

Figure 2 is a flow chart illustrating a second embodiment of the invention.

Detailed Description

Rather than simply counting the number of times a sequence of target instructions is interpreted before it is translated and optimized, the present invention attempts to lend some intelligence to the process. It does this by utilizing processes which attempt to maintain a balance selected by system designers of the amount of interpretation versus the amount of translation.

In a basic embodiment of the invention illustrated in Figure 1, the amount of time being spent in interpreting target instructions as contrasted to the amount of time spent in translating target instructions

is determined. If the time spent interpreting is too great, then the processor simply switches to translating. As the code morphing software is converting sequences of target instructions into sequences of host instructions by interpreting or translating and is executing those converted sequences, the process of the invention essentially tests the amount of time being spent in each of interpretation and translation processes in order to determine if too much time is being spent interpreting. The utilization of this new process allows the operations being conducted by the processor to determine whether the method of instruction conversion being used is optimal for accelerating the overall operation of the processor.

One embodiment for carrying out the testing operation utilizes a timer to select intervals at which the particular operation being run by the code morphing software is tested. For example, every thousandth of a second the operation may be tested to determine whether the interpreter or the translator is running. The test itself is statistically more likely to occur during periods in which excess time is being taken and thus during conversion by the interpreter. The result of each interrogation is furnished to a counter which counts up for each determination that the interpreter is running and down for each determination that the translator is running. If the count reaches a maximum, then the operation of the software is switched from interpreting to translating at the next execution of that sequence of instructions no matter what sequence of instructions is being converted. This helps to increase the amount of translation being conducted compared to interpretation and speeds up execution. On the other hand, as long as the counter does not

reach a maximum, the operation of generating host instructions from target instructions continues without change.

In general, this method of testing determines whether or not the software of the processor is doing too much interpreting as contrasted to translation. The maximum value at which the counter is set may be selected in accordance with the invention to reflect the amount of the interpreting operation determined to be desirable for the best utilization of the processor. For example, if more interpreting is desired, a higher maximum value is selected; while if more translating is desired, a lower maximum value is selected. It is also possible to weight the results of interrogation differently for interpretation and translation. For example, a test finding that interpretation is running might increase the count by one, while a test finding that translation is running might decrease the count by only one-half. Other values weighted to favor one or the other of the processes might also be used based on the results desired by system designers. This method of switching between interpretation and translation provides more accurate results than simply counting the number of times a particular operation is interpreted before switching to translation.

Another embodiment of the invention accomplishes the testing by utilizing an interval determined by the execution of a set number of instructions rather than an interval of time. This embodiment is more likely to provide accurate information regarding the percentage of time the interpreter is operating than the embodiment using a timed interval because it compares instructions to instructions rather than instructions to time. This embodiment requires using a single counter to measure the number of instructions executed by the host. After a set number of instructions

have been executed, the code morphing software tests the operating process to determine whether the interpreter or the translator is running. The remainder of the process is carried out in the same manner as the embodiment using a timed interval to determine when to test to determine whether the process is interpretation or translation.

One advanced embodiment of the invention shown in Figure 2 utilizes a combination of the counting method disclosed in the patent application described above and the method of testing at intervals to determine which process is being executed. Both the counting and interval testing methods are run constantly while the conversion process proceeds. The counting method counts the number of times each particular sequence of instructions is interpreted and switches to translation of the sequence when a specified number of interpretations of the sequence have executed without error. The counting scheme is able to determine very rapidly when short processes such as tight loops have been running a numbers of times and should be translated. The interval testing method runs at the same time to determine whether the overall operation of conversion is spending too much time interpreting. The interval testing method of determining when to switch between conversion processes is able to detect more rapidly when longer sequences of instructions are being interpreted than the counting method and thus switches to translation faster for such sequences.

More advanced embodiments of the present invention may utilize another combination of the original counting method and the interval testing method. More particularly, one such embodiment may provide for counting the number of times instructions are interpreted until the same instruction has been interpreted some selected number of times. During

this counting process, a count of the number of times the sequence has been interpreted is accomplished and statistics may be kept for each sequence. Once the maximum count has been reached, the code morphing software shifts to a translation process. During the same period, the code morphing software may continue testing at intervals the mode in which the host processor is converting all instructions to determine whether the conversion process should be switched from interpreting to translating. When either test is met, the code morphing software switches from interpreting to translating the particular sequence. However, the translation accomplished may be less than optimal and thus quickly completed. For example, the translation process might translate and reorder to favor a particular branch without any linking to other sequences. When executing the modestly translated process, the software may also use the counting test and gather branch statistics but over a longer period of time so that more knowledge of the actual operation of the translated sequence may be derived. These statistics are very useful in determining the form more optimized translations should take. Alternatively, or in addition, the code morphing software may continue testing at intervals the mode in which the host processor is converting all instructions to determine whether the translation should be switched from modest translation process to a more thorough translation process such as one involving significant amounts of linking of sequences and optimizing across linked sequences. If the testing process reaches a maximum at a test point at which a minimum translation is executing, the process switches to translation with more advanced optimization controlled by the much larger pool of accumulated branch statistics.

There are many possible modifications of the process of the present invention which can be implemented. Theoretically, there is no reason that any number of intermediate steps of translation cannot be implemented utilizing different levels during testing to determine whether to switch to a next level of translation.

Although the present invention has been described in terms of a preferred embodiment, it will be appreciated that various modifications and alterations might be made by those skilled in the art without departing from the spirit and scope of the invention. The invention should therefore be measured in terms of the claims which follow.

What Is Claimed Is:

Claim 1. A method of transferring between types of conversion processes in a computer which converts instructions from a target instruction set to a host instruction set comprising the steps of:

executing code morphing software including an interpreter and a translator to generate host instructions from target instructions,

detecting at intervals whether the interpreter or the translator is operating,

increasing a count if the interpreter is operating and decreasing the count if the translator is operating, and

changing from interpreting to translating a sequence of target instructions when the count reaches a selected maximum.

Claim 2 A method as claimed in Claim 1 in which the interval is a selected time period.

Claim 3. A method as claimed in Claim 1 in which the interval is a selected number of executed target instructions.

Claim 4. A method as claimed in Claim 1 in which the amount the count is increased at a detection of interpretation is selectable.

Claim 5. A method as claimed in Claim 1 in which the amount the count is decreased at a detection of translation is selectable.

Claim 6. A method as claimed in Claim 1 comprising the further steps of:

counting each instance in which a sequence of instructions is interpreted,

changing from interpreting to translating a sequence of target instructions when the count of instances reaches a selected maximum.

Claim 7. A method as claimed in Claim 7 comprising the further steps of:

gathering statistics regarding each sequence of instructions, and

optimizing translation of a sequence of instructions based on statistics gathered.

Claim 8. A method as claimed in Claim 1 in which the step of changing from interpreting to translating a sequence of target instructions when the count reaches a selected maximum includes translation with limited optimization, and

which further includes the steps of:

testing while executing a sequence of target instructions translated with limited optimization to determine whether the sequence should be further optimized, and

retranslating and further optimizing in response to the testing.

Claim 9. A method as claimed in Claim 8 in which the step of testing while executing a sequence of target instructions translated with limited optimization includes counting each instance in which a sequence of instructions is executed, and

the step of retranslating and further optimizing occurs when the count of instances reaches a selected maximum.

Claim 10. A method of optimizing execution by a computer which dynamically converts instructions from a target instruction set to a host instruction set comprising the steps of:

providing a plurality of instruction conversion processes each providing a different level of optimization for converted instructions from a target instruction set to a host instruction set,

providing means for determining dynamically which conversion process best converts each sequence of instructions, and

converting a sequence of instructions using a conversion process determined to best convert the sequence of instructions.

Claim 11. A method as claimed in Claim 10 in which the conversion processes include interpretation and translation.

Claim 12. A method as claimed in Claim 10 in which the conversion processes include interpretation, translation with minimal optimization, and translation with advanced optimization..

Claim 13. A method as claimed in Claim 10 in which the means for determining dynamically which conversion process best converts each sequence of instructions depends on the number of times each sequence is converted by a particular conversion process.

Claim 14. A method as claimed in Claim 10 in which the means for determining dynamically which conversion process best converts each sequence of instructions depends on a ratio of the number of times one conversion process is run compared to another conversion process.

Claim 15. A method as claimed in Claim 10 in which the means for determining dynamically which conversion process best converts each sequence of instructions

depends on the number of times each sequence is converted by a particular conversion process, and

depends on a ratio of the number of times one conversion process is run compared to another conversion process.

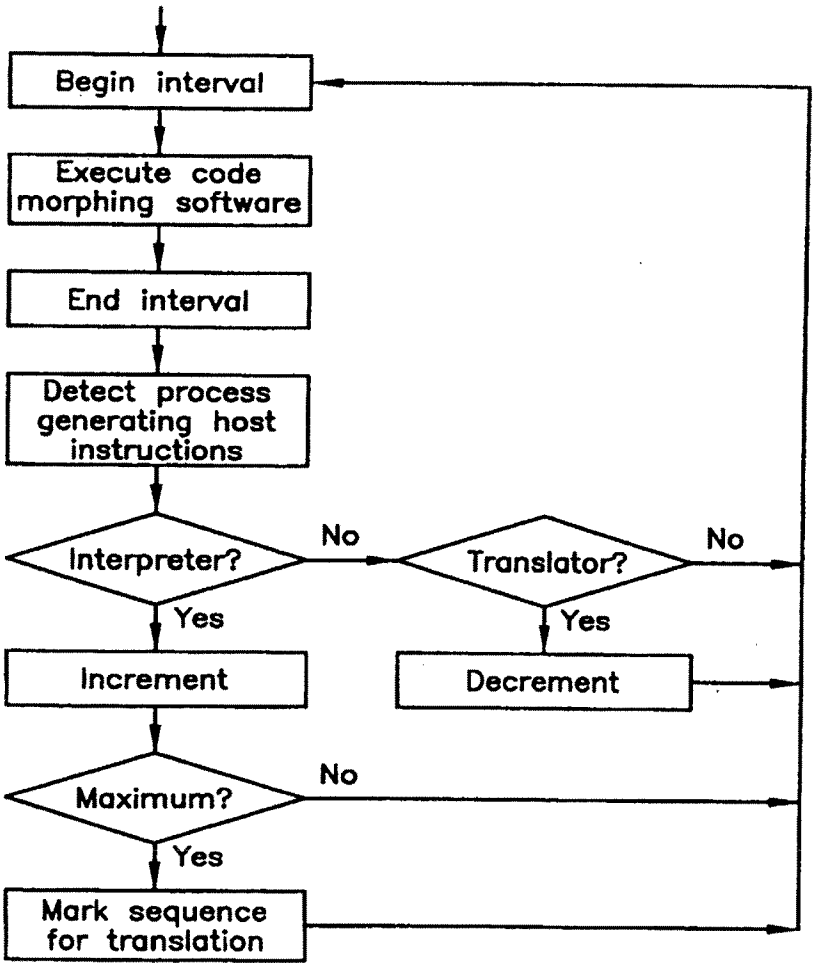

Begin interval
Execute code morphing software
End interval
Detect process generating host instructions
Interpreter?
No
Translator?
No
Yes
Yes
Increment
Decrement
Maximum?
No
Yes
Mark sequence for translation

Fig. 1

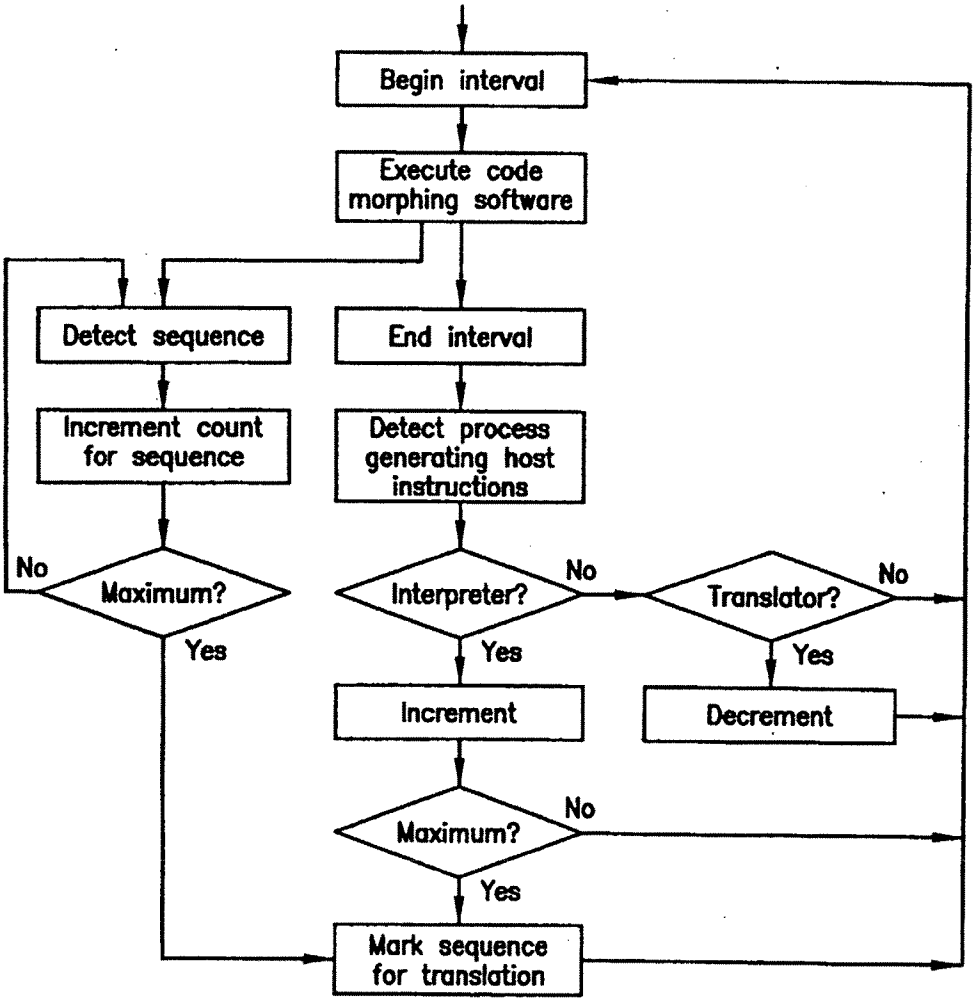

Begin interval
Execute code morphing software
Detect sequence
End interval
Increment count for sequence
Detect process generating host instructions
No
Maximum?
Yes
Interpreter?
No
Translator?
No
Yes
Yes
Increment
Decrement
Maximum?
No
Yes
Mark sequence for translation

Fig. 2